ulo-storage

Indexing and Querying Organizational Data in Mathematical Libraries

Andreas Schärtl
andreas.schaertl@fau.de

Organizational knowledge extracted from existing formal libraries has the potential to be usable in the design of a universal search engine for mathematical research. However, it is not enough to merely collect and export formal knowledge in a unified format, it is also necessary for this information to be available for querying.

ulo-storage aims to lay out the groundwork for this task. We collected various pieces of exported organizational knowledge into a centralized and efficient store, made the resulting knowledge graph available on the network and then evaluated different ways of querying that store. In addition, implementations of some exemplary queries on top of our created infrastructure resulted in insights on how unified schemas for organizational mathematical knowledge could be extended.

Erklärung. Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration. I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

Andreas Schärtl, Erlangen, 2020/11/13

Contents

1. Introduction **4**

2. Implementation **5**
2.1. Components Implemented for *ulo-storage* 5
2.2. Collector and Importer 6
2.3. Endpoint 9
2.4. Deployment and Availability 11
2.5. Summary 12

3. Applications **12**
3.1. Exploring Existing Data Sets 12
3.2. Interactive Exploration 14
3.3. Querying for Tetrapodal Search 16
3.4. Summary 20

4. Towards Manageable Ontologies **22**
4.1. The Challenge of Universality 22
4.2. A Layered Knowledge Architecture 22

5. Conclusion **23**

A. ULO Predicates Used in Coq and Isabelle Exports **25**

1. Introduction

To tackle the vast array of mathematical publications, various ways of *computerizing* mathematical knowledge have been experimented with. Already it is difficult for human mathematicians to keep even a subset of all mathematical knowledge in their mind, a problem referred to as the "one brain barrier" in literature [1]. A hope is that computerization will yield great improvement to formal research by making the results of all collected publications readily available and easy to search.

One topic of research in this field is the idea of a *tetrapodal search* that combines four distinct areas of mathematical knowledge. These four kinds being (1) the actual formulae as *symbolic knowledge*, (2) examples and concrete objects as *concrete knowledge*, (3) prose, names and comments as *narrative knowledge* and finally (4) identifiers, references and their relationships, referred to as *organizational knowledge* [2].

Tetrapodal search aims to provide a unified search engine that indexes each of these four different subsets of mathematical knowledge. Because all four kinds of knowledge are inherently unique in their structure, tetrapodal search proposes that each subset of mathematical knowledge should be made available in a storage backend that fits the kind of data it is providing. With all four areas available for querying, tetrapodal search intends to then combine the four indexes into a single query interface. Currently, research aims at providing schemas, storage backends and indexes for the four different kinds of mathematical knowledge. In this context, the focus of *ulo-storage* is the area of organizational knowledge.

A previously proposed way to structure organizational knowledge is the *upper level ontology* (ULO) [3]. ULO takes the form of an OWL ontology [4] and as such all organization knowledge is stored as RDF triplets with a unified schema of ULO predicates [5]. Some effort has been made to export existing collections of formal mathematical knowledge to ULO. In particular, exports from Isabelle and Coq-based libraries are available [6, 7, 8]. The resulting data set is already quite large, the Isabelle export alone containing more than 200 million triplets.

Existing exports from Isabelle and Coq result in sets of XML files that contain RDF triplets. This is a convenient format for exchange and easily tracked using version control systems such as Git [9] as employed by MathHub [10]. However, considering the vast number of triplets, it is impossible to query easily and efficiently in this state. This is what *ulo-storage* is focused on, that is making ULO data sets accessible for querying and analysis. We collected RDF files spread over different Git repositories, imported them into a database and then experimented with APIs for accessing that data set.

The contribution of *ulo-storage* is twofold. First, (1) we built up various infrastructure components that make organizational knowledge easy to query. These components can make up building blocks of a larger tetrapodal search system. Their design and implementation are discussed in Section 2. Second, (2) we ran prototype applications and queries on top of this infrastructure. While the applications are not useful in itself, they can give us insight about future development of the upper level ontology and related schemas. These applications and queries are the focus of Section 3. A summary of

encountered problems and suggestions for next steps concludes this report in Section 5.

2. Implementation

One of the two contributions of *ulo-storage* is that we implemented components for making organizational mathematical knowledge (formulated as RDF triplets) queryable. This section first makes out the individual components involved in this task. We then discuss the actual implementation created for this project.

2.1. Components Implemented for *ulo-storage*

Figure 1 illustrates how data flows through the different components. In total, we made out three components that make up the infrastructure provided by *ulo-storage*.

- ULO triplets are present in various locations, be it Git repositories, web servers or the local disk. It is the job of a *Collector* to assemble these RDF files and forward them for further processing. This may involve cloning a Git repository or crawling the file system.
- With streams of ULO files assembled by the Collector, these streams then get passed to an *Importer*. The Importer then uploads RDF streams into some kind of permanent storage. As we will see, the GraphDB [11] triple store was a natural fit.
- Finally, with all triplets stored in a database, an *Endpoint* is where applications access the underlying knowledge base. This does not necessarily need to be any custom software, rather the programming interface of the underlying database server itself can be understood as an Endpoint of its own.

Collector, Importer and Endpoint provide us with an automated way of making RDF files available for use within applications. We will now take a look at the actual implementation created for *ulo-storage*, beginning with the implementation of Collector and Importer.

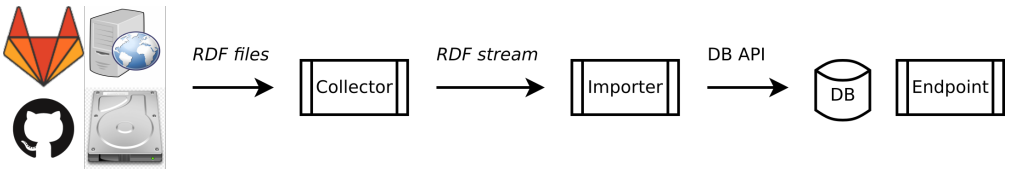


Figure 1: Components involved in the *ulo-storage* system.

2.2. Collector and Importer

We previously described Collector and Importer as two distinct components. First, a Collector pulls RDF data from various sources as an input and outputs a stream of standardized RDF data. Second, an Importer takes such a stream of RDF data and then dumps it to some sort of persistent storage. In our implementation, both Collector and Importer ended up as one piece of monolithic software. This does not need to be the case but proved convenient as combining Collector and Importer forgoes the needs for an additional IPC mechanism between Collector and Importer. In addition, neither our Collector nor Importer are particularly complicated pieces of software, as such there is no pressing need to force them into separate processes.

Our implementation supports two sources for RDF files, namely Git repositories and the local file system. The file system Collector crawls a given directory on the local machine and looks for RDF XML files [12] while the Git Collector first clones a Git repository and then passes the checked out working copy to the file system Collector. Because we found that is not uncommon for RDF files to be compressed, our implementation supports on the fly extraction of gzip [13] and xz [14] formats which can greatly reduce the required disk space in the collection step.

During development of the Collector, we found that existing exports from third party mathematical libraries contain RDF syntax errors which were not discovered previously. In particular, both Isabelle and Coq exports contained URIs which does not fit the official syntax specification [15] as they contained illegal characters. Previous work [3] that processed Coq and Isabelle exports used database software such as Virtuoso Open Source [16] which does not properly check URIs according to spec; in consequence these faults were only discovered now. To tackle these problems, we introduced on the fly correction steps during collection that escape the URIs in question and then continue processing. Of course this is only a work-around. Related bug reports were filed in the respective export projects to ensure that in the future this extra step is not necessary.

The output of the Collector is a stream of RDF data. This stream gets passed to the Importer which imports the encoded RDF triplets into some kind of persistent storage. In theory, multiple implementations of this Importer are possible, namely different implementations for different database backends. As we will see in Section 2.3, for our project we selected the GraphDB triple store alone. The Importer merely needs to make the necessary API calls to import the RDF stream into the database. As such the import itself is straight-forward, our software only needs to upload the RDF file stream as-is to an HTTP endpoint provided by our GraphDB instance.

To review, our combination of Collector and Importer fetches XML files from Git repositories, applies on the fly decompression and fixes and then imports the collected RDF triplets into persistent database storage.

2.2.1. Scheduling

Collector and Importer were implemented as library code that can be called from various front ends. For this project, we provide both a command line interface as well as a

graphical web front end. While the command line interface is only useful for manually starting single runs, the web interface (Figure 2) allows for more flexibility. In particular, import jobs can be started either manually or scheduled to run at fixed intervals. The web interface also persists error messages and logs.

2.2.2. Version Management

Automated job control leads us to the problem of versioning. In our current design, given ULO exports $\mathcal{E}_i$ depend on original third party libraries $\mathcal{L}_i$. Running $\mathcal{E}_i$ through the workflow of Collector and Importer, we get some database representation $\mathcal{D}$. We see that data flows

$$\begin{aligned} \mathcal{L}_1 &\to \mathcal{E}_1 \to \mathcal{D} \\ \mathcal{L}_2 &\to \mathcal{E}_2 \to \mathcal{D} \\ &\vdots \\ \mathcal{L}_n &\to \mathcal{E}_n \to \mathcal{D} \end{aligned}$$

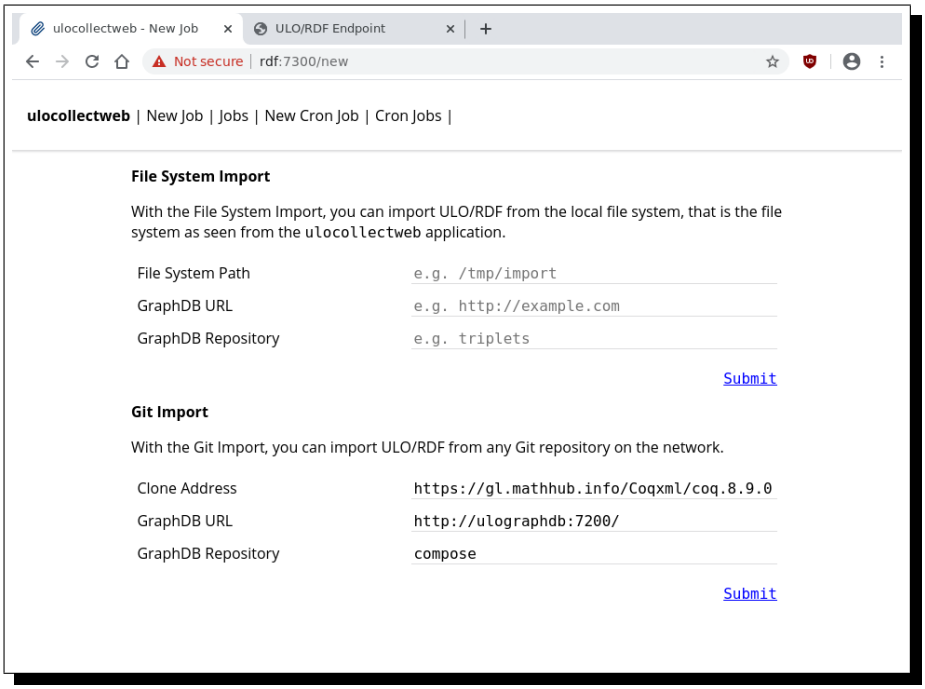
from n individual libraries $\mathcal{L}_i$ into a single database storage $\mathcal{D}$ that is used for querying.

However, we must not ignore that mathematical knowledge is ever changing and not static. When a given library $\mathcal{L}_i^t$ at revision t gets updated to a new version $\mathcal{L}_i^{t+1}$, this change will eventually propagate to the associated export and result in a new set of RDF triplets $\mathcal{E}_i^{t+1}$. Our global database state $\mathcal{D}$ needs to get updated to match the changes between $\mathcal{E}_i^t$ and $\mathcal{E}_i^{t+1}$.

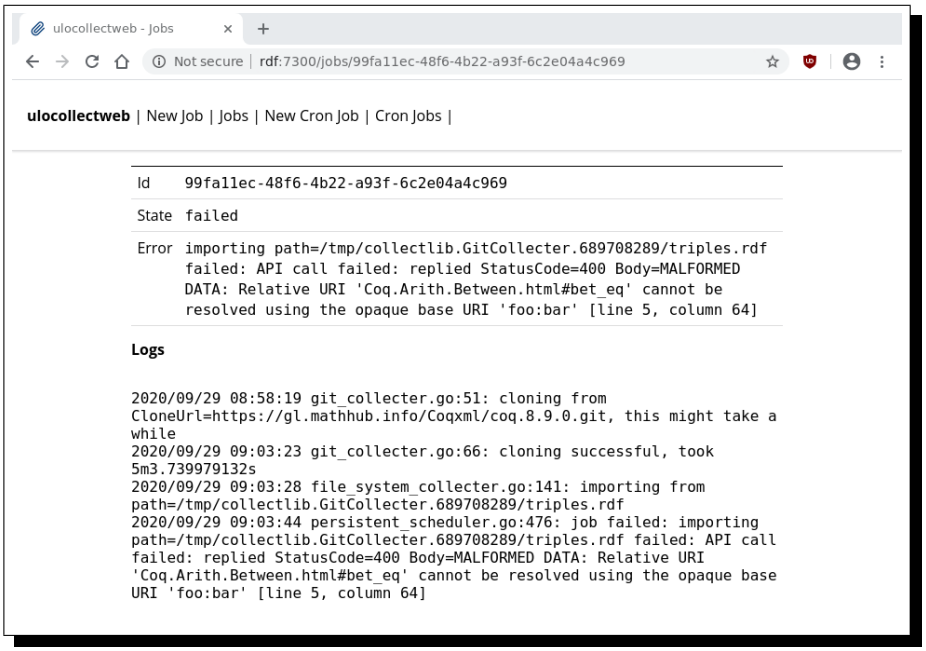
Finding an efficient implementation for this problem is not trivial. While it should be possible to compute the difference between two exports $\mathcal{E}_i^t$ and $\mathcal{E}_i^{t+1}$ and infer the changes necessary to be applied to $\mathcal{D}$, the big number of triplets makes this appear unfeasible. As this is a problem an implementer of a greater tetrapodal search system will most likely encounter, we suggest the following approaches to tackle this challenge.

One approach is to annotate each triplet in $\mathcal{D}$ with versioning information about which particular export $\mathcal{E}_i^t$ it was derived from. During an import from $\mathcal{E}_i^s$ into $\mathcal{D}$, we could (1) first remove all triplets in $\mathcal{D}$ that were derived from previous version $\mathcal{E}_i^{s-1}$ and (2) then re-import all triplets from the current version $\mathcal{E}_i^s$. Annotating triplets with versioning information is an approach that should work, but it does introduce $\mathcal{O}(n)$ additional triplets in $\mathcal{D}$ where n is the number of triplets in $\mathcal{D}$. After all, we need to annotate each of the n triplets with versioning information, effectively doubling the required storage space. A not very satisfying solution.

Another approach is to regularly re-create the full data set $\mathcal{D}$ from scratch, say every seven days. This circumvents the problems related to updating existing data sets, but also means that changes in a given library $\mathcal{L}_i$ take some to propagate to $\mathcal{D}$. Continuing this train of thought, an advanced version of this approach could forgo the requirement for one single database storage $\mathcal{D}$ entirely. Instead of maintaining just one global database state $\mathcal{D}$, we suggest experimenting with dedicated database instances $\mathcal{D}_i$ for each given library $\mathcal{L}_i$. The advantage here is that re-creating a given database representation $\mathcal{D}_i$ is fast as exports $\mathcal{E}_i$ are comparably small. The disadvantage is that we



(a) Scheduling a new import job in the web interface.



(b) Reviewing a previous import job. This particular job failed, which illustrates how our web interface presents errors and logs.

Figure 2: The web interface for controlling the Collector and Importer components. While Collector and Importer can also be used as library code or with a command line utility, managing jobs in a web interface is probably the most convenient.

still want to query the whole data set $\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2 \cup \cdots \cup \mathcal{D}_n$. This does require the development of some cross-database query mechanism, functionality existing systems currently only offer limited support for [17].

In summary, we see that versioning is a potential challenge for a greater tetrapodal search system. While not a pressing issue for *ulo-storage* now, we consider it a topic of future research.

2.3. Endpoint

Finally, we need to discuss how *ulo-storage* realizes the Endpoint. Recall that an Endpoint provides the programming interface for applications that wish to query our collection of organizational knowledge. In practice, the choice of Endpoint programming interface is determined by the choice of database system as the Endpoint is provided directly by the database system.

In our project, organizational knowledge is formulated as RDF triplets. The canonical choice for us is to use a triple store, that is a database optimized for storing RDF triplets [18, 19]. For our project, we used the GraphDB [11] triple store. A free version that fits our needs is available at [20].

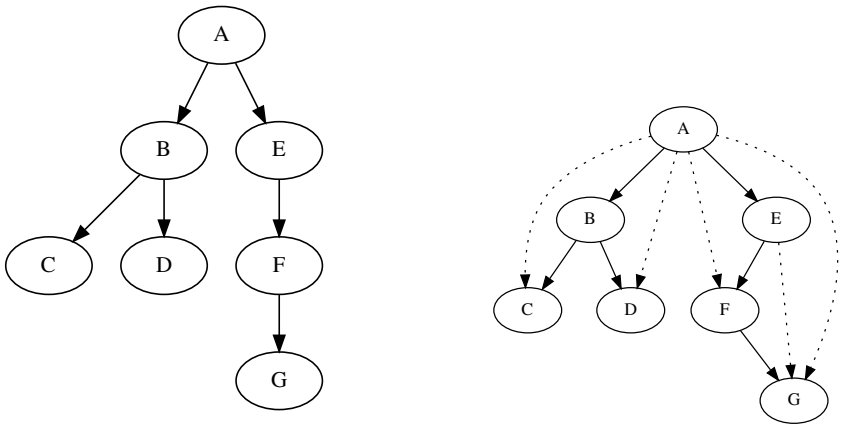
2.3.1. Transitive Queries

A notable advantage of GraphDB compared to other systems such as Virtuoso Open Source [16, 3] is that GraphDB supports recent versions of the SPARQL query language [21] and OWL Reasoning [22, 23]. In particular, this means that GraphDB offers support for transitive queries as described in previous work on ULO [3]. A transitive query is one that, given a relation R, asks for the transitive closure $S$of R. (Figure 3).

In fact, GraphDB supports two approaches for realizing transitive queries. On one hand, GraphDB supports the `owl:TransitiveProperty` [22, Section 4.4.1] property that defines a given predicate P to be transitive. With P marked this way, querying the knowledge base is equivalent to querying the transitive closure of P. This requires transitivity to be hard-coded into the knowledge base. If we only wish to query the transitive closure for a given query, we can take advantage of so-called "property paths" [25] which allow us to indicate that a given predicate P is to be understood as transitive when querying. Only during querying is the transitive closure then evaluated. Either way, GraphDB supports transitive queries without awkward workarounds necessary in other systems [3].

2.3.2. SPARQL Endpoint

There are multiple approaches to querying the GraphDB triple store, one based around the standardized SPARQL query language and the other on the RDF4J Java library. Both approaches have unique advantages. Let us first take a look at SPARQL, which is a standardized query language for RDF triplet data [26]. The specification includes



(a) We can think of this tree as visualizing a relation R where $(X, Y) \in R$ iff there is an edge from X to Y.

(b) Transitive closure S of relation R. Additionally to each tuple from R (solid edges), S also contains additional transitive (dotted) edges.

Figure 3: Illustrating the idea behind transitive closures. A transitive closure S of relation R is defined as the "minimal transitive relation that contains R" [24].

not just syntax and semantics of the language itself, but also a standardized REST interface [27] for querying database servers.

The SPARQL syntax was inspired by SQL and as such the `SELECT WHERE` syntax should be familiar to many software developers. A simple query that returns all triplets in the store looks like

```
SELECT * WHERE { ?s ?p ?o }
```

where `?s`, `?p` and `?o` are query variables. The result of any query are valid substitutions for all query variables. In this particular case, the database would return a table of all triplets in the store sorted by subject `?o`, predicate `?p` and object `?o`.

Probably the biggest advantage is that SPARQL is ubiquitous. As it is the de facto standard for querying triple stores, lots of implementations (client and server) as well as documentation are available [28, 29, 30].

2.3.3. RDF4J Endpoint

SPARQL is one way of accessing a triple store database. Another approach is RDF4J, a Java API for interacting with RDF graphs, implemented based on a superset of the SPARQL REST interface [31]. GraphDB is one of the database servers that supports RDF4J, in fact it is the recommended way of interacting with GraphDB repositories [32].

Instead of formulating textual queries, RDF4J allows developers to query a knowledge

base by calling Java library methods. Previous query that asks for all triplets in the store looks like

```
connection.getStatements(null, null, null);
```

in RDF4J. `getStatements(s, p, o)` returns all triplets that have matching subject `s`, predicate `p` and object `o`. Any argument that is `null` can be substituted with any value, that is it is a query variable to be filled by the call to `getStatements`.

Using RDF4J does introduce a dependency on the JVM and its languages. But in practice, we found RDF4J to be quite convenient, especially for simple queries, as it allows us to formulate everything in a single programming language rather than mixing programming language with awkward query strings. We also found it quite helpful to generate Java classes from OWL ontologies that contain all definitions of the ontology as easily accessible constants [33]. This provides us with powerful IDE auto completion features during development of ULO applications.

Summarizing the last two sections, we see that both SPARQL and RDF4J have unique advantages. While SPARQL is an official W3C [34] standard and implemented by more database systems, RDF4J can be more convenient when dealing with JVM-based projects. For *ulo-storage*, we played around with both interfaces and chose whatever seemed more convenient at the moment. We recommend any implementors to do the same.

2.4. Deployment and Availability

Software not only needs to get developed, but also deployed. To deploy the combination of Collector, Importer and Endpoint, we use Docker Compose. Docker itself is a technology for wrapping software into containers, that is lightweight virtual machines with a fixed environment for running a given application [35, pp. 22]. Docker Compose then is a way of combining individual Docker containers to run a full tech stack of application, database server and so on [35, pp. 42]. All configuration of the overarching setup is stored in a Docker Compose file that describes the software stack.

For *ulo-storage*, we provide a single Docker Compose file which starts three containers, namely (1) the Collector/Importer web interface, (2) a GraphDB instance which provides us with the required Endpoint and (3) some test applications that use that Endpoint. All code for Collector and Importer is available in the `ulo-storage-collect` Git repository [36]. Additional deployment files, that is Docker Compose configuration and additional Dockerfiles are stored in a separate repository [37].

2.5. Summary

With this, we conclude our discussion of the implementation developed for the *ulo-storage* project. We designed a system based around (1) a Collector which collects RDF triplets from third party sources, (2) an Importer which imports these triplets into a GraphDB database and (3) looked at different ways of querying a GraphDB Endpoint. All of this is easy to deploy using a single Docker Compose file.

Our concrete implementation is useful in so far as that we can use it to experiment with ULO data sets. But development also provided insight about (1) which components this class of system requires and (2) which problems need to be solved. One topic we discussed at length is version management. It is easy to dismiss this in these early stages of development, but no question it is something to keep in mind.

3. Applications

With programming endpoints in place, we can now query the data set containing both Isabelle and Col exports stored in GraphDB. We experimented with the following applications that talk to a GraphDB Endpoint, our second contribution.

- Exploring which ULO predicates are actually used in the existing Coq and Isabelle exports. We find that more than two thirds of existing ULO predicates were not taken advantage of (Section 3.1).
- Providing an interactive interface for browsing the data set. Our implementation is limited to listing basic information about contributors and their work (Section 3.2).
- We investigated queries that could be used to extend the system into a larger tetrapodal search system. While some organizational queries have obvious canonical solutions, others introduce questions on how organizational knowledge should be structured (Section 3.3).

We will now discuss each application in more detail.

3.1. Exploring Existing Data Sets

For our first application, we looked at which ULO predicates are actually used by the respective data sets. With more than 250 million triplets in the store, we hoped that this would give us some insight into the kind of knowledge we are dealing with.

Implementing a query for this job is not very difficult. In SPARQL, this can be achieved with the `COUNT` aggregate, the full query is given in verbatim in Figure 4a. Our query yields a list of all used predicates together with the number of occurrences (Figure 4b). Looking at the results, we find that both the Isabelle and the Coq data sets only use subsets of the predicates provided by ULO. The full results are listed in Appendix A. In both cases, what stands out is that either export uses less than a third of all available ULO predicates.

We also see that the Isabelle and Coq exports use different predicates. For example, the Isabelle export contains organizational meta information such as information about paragraphs and sections in the source document while the Coq export only tells us the filename of the original Coq source. Both exports have their particularities and with more and more third party libraries exported to ULO, one has to assume that this heterogeneity will only grow. In particular we want to point to the large number of

```
PREFIX ulo: <https://mathhub.info/ulo#>

SELECT ?predicate (COUNT(?predicate) as ?count)
WHERE {
    ?s ?predicate ?o .
}
GROUP BY ?predicate
ORDER BY DESC(?count)
```

(a) SPARQL query that returns a list of all `predicate`s used in the backing store. We include the `ulo` prefix such that the results are printed in a concise human readable format.

	predicate	count
1	ulo:uses	1160140
2	ulo:declares	88862
3	ulo:internal-size	82336
4	ulo:derived	69017
5	ulo:statement	44638
6	ulo:object	15179
7	ulo:primitive	14459
8	ulo:proposition	10437
⋮	⋮	⋮

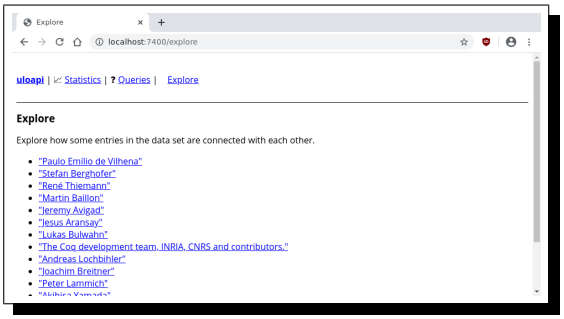
(b) Result of query from Figure 4a. The database returns a list of all involved `predicate`s ordered by their `count`. The particular results here are from the core Coq export [38].

Figure 4: Querying the predicates in use in a given data set.

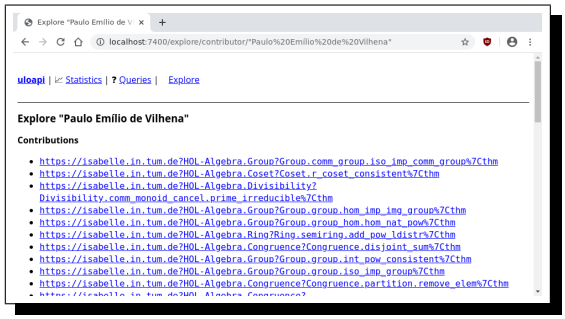
predicates which remain unused in both Isabelle and Coq exports. A user formulating queries for ULO might be oblivious to the fact that only subsets of exports support given predicates.

We expect the difference between Coq and Isabelle exports to be caused by the difference in source material. It is only natural that different third party libraries expressed in different languages with different capabilities will result in different ULO predicates. Regardless, we want to hint at the possibility that this could also be an omission in the exporter code that originally generated the RDF triplets. This shows the importance of writing good exporters. Exporters translating existing libraries to ULO triplets must lose as little information as possible to ensure good results in a larger tetrapodal search system.

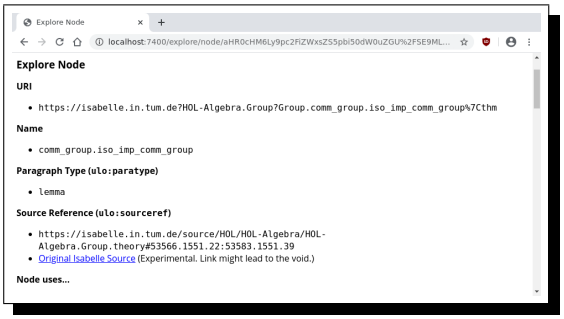
Our first application gave us an initial impression of the structure of currently available



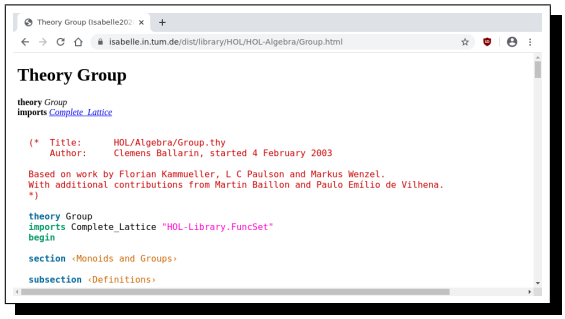
(a) Listing all contributors in the data set. Clicking on a given author shows their individual contributions.



(b) Listing works a given contributor has worked on.



(c) Examining a single node in the data set. From `ulo:sourceref` we can infer the original URI of this object.



(d) The original source code for a given node in our data set.

Figure 5: Exploring the Isabelle export [7] with a rudimentary web interface. A demo of this tool is available at `https://ulordf4j.mathhub.info`.

organizational knowledge formulated in ULO triplets. Whether caused by the difference in formal language or because of omissions in code that produces ULO triplets, we must not expect predicates to be evenly distributed in the data set. This is something to keep in mind, especially as the number of ULO exports increases.

3.2. Interactive Exploration

The second application we want to discuss illustrates how to browse an ULO data set interactively. Here, we developed a web front end that allows users to browse contributions to the knowledge base sorted by author. Figure 5 shows four screenshots of the current version, a public demo is available at `https://ulordf4j.mathhub.info`.

In this particular case, we used the RDF4J Endpoint, the application itself is implemented in Java. Once the user has selected a given contribution, we list some basic information about the selected object, such as type (e.g. lemma or corollary) and name as well as references to other objects in the knowledge base.

This experiment is interesting because similar approaches could be applied when designing IDE integration for working with organizational knowledge. Given some object in a document identified by an URI, we can look at how that object is connected to other objects in the knowledge graph.

Our front end can also connect the dots back from ULO object to original source code. In particular, we took a look at the `ulo:sourceref` predicate defined for many objects in the knowledge base. `ulo:sourceref` is supposed to contain "the URI of the physical location (e.g., file/URI, line, column) of the source code that introduced the subject" [4]. But while making the connection from exported ULO object to original Isabelle document is convenient for the user, this feature required some extra work from us as application implementors. The reason for this is that the format of the `ulo:sourceref` property appears to be not well-defined, rather it is up to implementors of library exporters how to format these source references. For example, in the Isabelle export [7], we have to translate source references such as

```
https://isabelle.in.tum.de/source/HOL/HOL-Algebra/
  HOL-Algebra.Group.theory#17829.576.8:17836.576.15
```

to the original source

```
https://isabelle.in.tum.de/dist/library/HOL/HOL-Algebra/Group.html
```

The consequence is that for each export, developers of front ends will need to write custom code for finding the original source. Maybe it would be a reasonable requirement for `ulo:sourceref` to have a well-defined format, ideally an actually reachable URI on the open web, if that is applicable for a given library.

While translating from `ulo:sourceref` to original URI introduced some extra work, implementing this application was straight-forward. With this we showed that searching through metadata in existing ULO data sets is already quite feasible. Implementing similar features for other environments should not be very difficult.

3.3. Querying for Tetrapodal Search

Various queries for a tetrapodal search system were previously suggested in literature [2]. We will now investigate how three of them could be realized with the help of ULO data sets and where other data sources are required. Where possible, we construct proof of concept implementations and evaluate their applicability.

3.3.1. Elementary Proofs and Computed Scores

We start with query $\mathcal{Q}_1$ which illustrates how we can compute arithmetic scores for objects in our knowledge graph. Query $\mathcal{Q}_1$ asks us to "*[f]ind theorems with non-elementary proofs*" [2]. Elementary proofs can be understood as those proof that are considered easy and obvious [39, 40]. In consequence, $\mathcal{Q}_1$ has to search for all proofs which are not trivial. Of course, just like any distinction between "theorem" and "corollary" is going to be somewhat arbitrary, so is any judgment about whether a proof is easy or not. While we

do understand elementary as easy here, we must not omit that there also exist concrete definitions of "elementary proofs" in certain subsets of mathematics [39, 41, 42]. As we will see, our naive understanding of elementary proofs results in interesting insights regardless.

Existing research on proof difficulty is either very broad or specific to one problem. For example, some experiments showed that students and prospective school teachers had problems with notation and term rewriting and were missing required prerequisites [43, 44], none of which seems applicable for grading individual formal proofs for difficulty. On the other hand, there is very specific research on rating proofs for concrete problems such as the satisfiability of a given CNF formula [45].

Organizational Aspect. A working hypothesis might be to assume that elementary proofs are short. In that case, the size, that is the number of bytes to store the proof, is our first indicator of proof complexity. This is by no means perfect, as even identical proofs can be represented in different ways that have vastly different size in bytes. It might be tempting to imagine a unique normal form for each proof, but finding such a normal form may very well be impossible. As it is very difficult to find a generalized definition of proof difficulty, we will accept proof size as a first working hypothesis.

ULO offers the `ulo:external-size` predicate which will allow us to sort by file size. Maybe proof complexity also leads to quick check times in proof assistants and automatic theorem provers. With this assumption in mind we could use the `ulo:check-time` predicate. Correlating proof complexity with file size or check time allows us to define one indicator of proof complexity based on organizational knowledge alone.

Other Aspects. A tetrapodal search system should probably also take symbolic knowledge into account. Based on some kind of measure of formula complexity, different proofs could be rated. Similarly, with narrative knowledge available to us, we could count the number of words, citations and so on to rate the narrative complexity of a proof. Combining symbolic knowledge, narrative knowledge and organizational knowledge should allow us to find proofs which are easier than others.

Implementation. Implementing a naive version of the organizational aspect can be as simple as querying for all theorems justified by proofs, ordered by size (or check time). Figure 6a illustrates how this can be achieved with a SPARQL query. But maybe we wish to go one step further and calculate a rating that assigns each proof some numeric score of complexity based on a number of properties. We can achieve this in SPARQL as recent versions support arithmetic as part of the SPARQL specification; Figure 6b shows an example. Finding a reasonable rating is its own topic of research, but we see that as long as it is based on standard arithmetic, it will be possible to formulate in a SPARQL query.

The queries in Figure 6 return a list of all theorems and associated proofs. Naturally, this list is bound to be very long. A suggested way to solve this problem is to introduce some kind of cutoff value for our complexity score. Another potential solution is to only list the first n results, something a user interface would have to do either way (i.e. pagination [46]). Either way, this is not so much an issue for the organizational storage engine and more one that a tetrapodal search aggregator has to account for.

```
PREFIX ulo: <https://mathhub.info/ulo#>

SELECT ?theorem ?proof ?size
WHERE {
    ?theorem ulo:theorem ?i .
    ?proof ulo:justifies ?theorem .
    ?proof ulo:proof ?j .
    ?proof ulo:external-size ?size .
}
ORDER BY DESC(?size)
```

(a) SPARQL query that returns each pair of `theorem` and one given `proof`, ordered by the size of `proof`.

```
PREFIX ulo: <https://mathhub.info/ulo#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?theorem ?proof ?size (
    xsd:float(?size) + xsd:float(?checktime) as ?
       rating
)
WHERE {
    ?theorem ulo:theorem ?i .
    ?proof ulo:justifies ?theorem .
    ?proof ulo:proof ?j .
    ?proof ulo:external-size ?size .
    ?proof ulo:check-time ?checktime .
}
ORDER BY DESC(?rating)
```

(b) SPARQL query that returns each pair of `theorem` and one given `proof` ordered by the sum of size and check time of the given proof. A naive rating such as this one will hardly yield helpful results, but it shows that given some kind of numeric complexity scale, we can rate theorems and their proofs by such a scale.

Figure 6: SPARQL queries for serving the organizational aspect of $\mathcal{Q}_1$.

Another problem is that computing these scores can be quite time intensive. Even if calculating a score for one given object is fast, doing so for the whole data set might quickly turn into a problem. In particular, if we wish to show the n objects with best scores, we do need to compute scores for all relevant objects first. In *ulo-storage*, all scores we experimented with were easy enough and the data sets small enough such that this did not become a concrete problem. But in a larger tetrapodal search system, caching results that were computed lazily or ahead of time will probably be a necessity. Which component takes care of keeping this cache is not clear right now, but we advocate for keeping caches of previously computed scores separate from the core *ulo-storage* Endpoint such that the Endpoint can be easily updated.

Understanding query $\mathcal{Q}_1$ in the way we did makes it difficult to present a definite solution. However while thinking about $\mathcal{Q}_1$ we found out that the infrastructure provided by *ulo-storage* allows us to compute arbitrary arithmetic scores, something that will surely be useful for many applications.

3.3.2. Categorizing Algorithms and Algorithmic Problems

The second query $\mathcal{Q}_2$ we decided to focus on wants to "*[f]ind algorithms that solve NP-complete graph problems*" [2]. Here we want the tetrapodal search system to return a listing of algorithms that solve (graph) problems with a given property (runtime complexity). We need to consider where and how each of these components might be found.

Symbolic and Concrete Aspects. First, let us consider algorithms. Algorithms can be formulated as computer code which can be understood as symbolic knowledge (code represented as a syntax tree) or as concrete knowledge (code as text files) [2, pp. 8–9, 47]. Either way, we will not be able to query these indices for what problem a given algorithm is solving, nor is it possible to infer properties as complex as NP-completeness automatically in the general case [48]. Metadata of this kind needs to be stored in a separate index for organizational knowledge, it being the only fit.

Organizational Aspect. If we wish to look up properties about algorithms from organizational knowledge, we first have to think about how to represent this information. As a first approach, we can try to represent algorithms and problems in terms of existing ULO predicates. As ULO does have a concept of `ulo:theorem` and `ulo:proof`, it might be tempting to exploit these predicates and represent algorithms understood as programs in terms of proofs. But that does not really capture the canonical understanding of algorithms. Algorithms are not actually programs, rather there are programs that implement algorithms. Even if we do work around this problem, it is not clear how to represent problems (e.g. the traveling salesman problem or the sorting problem) in terms of theorems (propositions, types) that get implemented by a proof (algorithm, program).

As algorithms make up an important part of certain areas of research, it might be reasonable to introduce native level support for algorithms in ULO or separately in another ontology. An argument for adding support directly to ULO is that ULO aims to be universal and as such should not be without algorithms. An argument for a separate ontology is that what we understand as ULO data sets (Isabelle, Coq exports)

already contain predicates from other ontologies (e.g. Dublin Core metadata [49, 50]). In addition, keeping concepts separated in distinct ontologies is not entirely unattractive in itself as it can be less overwhelming for a user working with these ontologies.

In summary, we see that it is difficult to service $\mathcal{Q}_2$ even though the nature of this query is very much one of organizational knowledge. It is probably up to the implementors of future ULO exports to find a good way of encoding algorithmic problems and solutions. Perhaps a starting point on this endeavor would be to find a formal way of structuring information gathered on sites like Rosetta Code [51], a site that provides concrete programs that solve algorithmic problems.

3.3.3. Contributors and Number of References

Finally, query $\mathcal{Q}_3$ from literature wants to know "*[a]ll areas of math that Nicolas G. de Bruijn has worked in and his main contributions*" [2]. $\mathcal{Q}_3$ is asking for works of a given author A. It also asks for their main contributions, for example which particularly interesting paragraphs or code A has authored. We picked this particular query as it is asking for metadata, something that should be easily serviced by organizational knowledge.

Organizational Aspect. ULO has no concept of authors, contributors, dates and so on. Rather, the idea is to take advantage of the Dublin Core project which provides an ontology for such metadata [49, 50]. For example, Dublin Core provides us with the `dcterms:creator` and `dcterms:contributor` predicates. Servicing $\mathcal{Q}_3$ requires us to look for creator A and then list all associated objects that they have worked on. Of course this requires above authorship predicates to actually be in use. With the Isabelle and Coq exports this was hardly the case; running some experiments we found less than 15 unique contributors and creators, raising suspicion that metadata is missing in the original library files. Regardless, existing ULO exports allow us to query for objects ordered by authors.

Implementation. A search for contributions by a given author can easily be formulated in SPARQL (Figure 7a). Query $\mathcal{Q}_3$ is also asking for the main contributions of A, that is those works that A authored that are the most important. Sorting the result by number of references can be a good start. To get the main contributions, we rate each individual work by its number of `ulo:uses` references. Extending the previous SPARQL query, we can ask the database for an ordered list of works, starting with the one that has the most references (Figure 7b). We see that one can formulate $\mathcal{Q}_3$ with just one SPARQL query. Because everything is handled by the database, access should be about as quick as we can hope it to be.

While the sparse data set available to use only returned a handful of results, we showed that queries such as $\mathcal{Q}_3$ are easily serviced with organizational knowledge formulated in ULO triplets. More advanced queries could look at the interlinks between authors and even uncover "citation cartels" as was done previously with similar approaches [52].

```
PREFIX ulo: <https://mathhub.info/ulo#>
PREFIX dcterms: <http://purl.org/dc/terms/>

SELECT ?work
WHERE {
    ?work dcterms:creator|dcterms:contributor "
       John Smith" .
}
GROUP BY ?work
```

(a) SPARQL query that asks for all `works` created by an author named "John Smith". ULO does not come with predicates for creator or contributor, instead the available data sets take advantage of the `dcterms` namespace [50].

```
PREFIX ulo: <https://mathhub.info/ulo#>
PREFIX dcterms: <http://purl.org/dc/terms/>

SELECT ?work (COUNT(?user) as ?refcount)
WHERE {
    ?work dcterms:creator|dcterms:contributor "
       John Smith" .
    ?user ulo:uses ?work .
}
GROUP BY ?work
ORDER BY DESC(?refcount)
```

(b) An adapted SPARQL query based on 7a. It lists all works authored by "John Smith" rated by number of references. The idea is that works that were referenced more often are more important.

	work	refcount
1	https://isabelle.in.tum.de?HOL-Algebra.Group?...	17
2	https://isabelle.in.tum.de?HOL-Algebra.Group?...	17
3	https://isabelle.in.tum.de?HOL-Algebra.Group?...	11
⋮	⋮	⋮

(c) Result of query from Figure 7b. In this particular case we asked for all contributions from "Paulo Emílio de Vilhena" in the Isabelle AFP export [7]. Our database Endpoint returns a listing of `works` sorted by their `refcount`. It would be the job of a tetrapodal search interface to translate these export-specific URIs into accessible references into the original source documents.

Figure 7: SPARQL queries for answering questions about authorship and main contributions.

3.4. Summary

Experimenting with $\mathcal{Q}_1$ to $\mathcal{Q}_3$ provided us with some insight into ULO and existing ULO exports. $\mathcal{Q}_1$ shows that while there is no universal definition for "elementary proof", ULO allows us to query for heuristics and calculate arbitrary arithmetic scores for objects of organizational knowledge. Query $\mathcal{Q}_2$ illustrates the difficulty in finding universal schemas. It remains an open question whether ULO should include algorithms as a first class citizen, as a concept based around existing ULO predicates or whether it is a better idea to design a dedicated ontology and data store entirely. Finally, while we were able to formulate a SPARQL query that should take care of most of $\mathcal{Q}_3$, we found that the existing data sets contain very little information about authorship. This underlines the observations made previously in Section 3.1 and should be on the mind of anyone writing exporters that output ULO triplets.

4. Towards Manageable Ontologies

Before finishing up this report with a general conclusion, we want to first dedicate a section to thoughts on the upper level ontology and ontology design in general. The contribution of this section is primarily of of potential for future work. At this point in time, the ideas formulated here lack concrete implementations.

4.1. The Challenge of Universality

ULO aims to be a universal language for capturing organizational mathematical knowledge. An outstandingly difficult task. ULO is aiming at nothing less than a universal schema on top of all collected (organizational) mathematical knowledge.

The current version of ULO already yields worthwhile results when formal libraries are exported to ULO triplets. Especially when it comes to metadata, querying such data sets proved to be easy. But an ontology such as ULO can only be a real game changer when it is truly universal, that is, when it is easy to formulate any kind of organizational knowledge in the form of a ULO data set.

As such it should not be terribly surprising that ULO forgoes the requirement of being absolutely correct. For example, what a `ulo:theorem` actually represent can differ depending on where the mathematical knowledge was originally extracted from. While at first this might feel a bit unsatisfying, it is important to realize that the strength of ULO data sets must be search and discovery. Particularities about meaning will eventually need to be resolved by more concrete and specific systems.

While that is not the hardest pill to swallow, it would be preferable to maintain organizational knowledge in a format that is both (1) as correct as possible and (2) easy to generalize and search. Future development of the upper level ontology first needs to be very clear on where it wants to position itself on this spectrum between accuracy and generalizability. In its position as an upper level ontology, we believe that ULO is best positioned as an ontology that favors generality at the cost of accuracy. It can serve

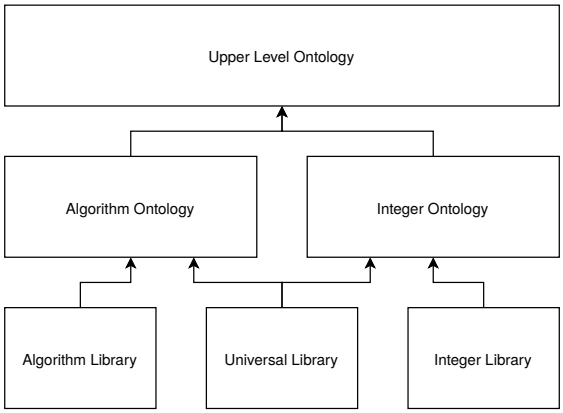


Figure 8: The idea behind layered ontologies is that instead of translating directly from formal library to ULO triplets, we would translate into intermediate ontologies specific to a given domain. These individual ontologies could then be compiled to ULO in an additional step.

as a generalized way of indexing vast amounts of formal knowledge, making it easy to discover and connect.

4.2. A Layered Knowledge Architecture

ULO as one concrete ontology will need to converge on one specific point on the accuracy-generalizability spectrum, namely at the place where generalizability is chosen in favor of accuracy. But this does not mean that we need to give up on accuracy as a whole. We believe that we can have both, we can have our cake and eat it it too.

Current exports investigated in this report take the approach of taking some library of formal knowledge and then converting that library directly into ULO triplets. Perhaps a better approach would be to use a layered architecture instead. The idea is sketched out in Figure 8. In this layered architecture, we would first convert a given third party library into triplets defined by an intermediate ontology. These triplets could then be compiled to ULO triplets for search. It is an approach not unlike intermediate byte codes used in compiler construction [53, pp. 357]. While lower layers preserve more detail, higher levels are more general and easier to search.

A valid criticism to this would be that we can understand the base library as an ontology of its own. In practice, the only difference is the file format. While formal libraries are formulated in some domain specific formal language, when we talk about ontologies, our understanding is that of OWL ontologies, that is RDF predicates with which knowledge is formulated. But RDF is easier to index using triple store databases such as GraphDB. And it should be easier to architecture a search system based around a unified format (RDF) rather than a zoo of formats and languages.

But a final judgment requires further investigation. Either way, we find it is necessary

to take the accuracy-generalizability spectrum into account and investigate how this spectrum can be serviced with different layers of ontologies.

5. Conclusion

Using the *ulo-storage* software stack introduced in Section 2 we were able to take existing RDF exports and import them into a GraphDB database. This made it possible to experiment with the applications and examples of Section 3. We showed that organizational knowledge formulated as ULO triplets can already give some insights. In particular, it is possible to formulate queries about meta information such as authorship and contribution. We also manged to resolve the interlinks between proofs and theorems.

Despite many remaining open questions, *ulo-storage* provides the necessary infrastructure for importing ULO triplets into an efficient storage engine. A necessary building block for a larger tetrapodal search system. In addition to the concrete implementation, the experiences we have made along the way should benefit future research towards a greater tetrapodal search system.

A. ULO Predicates Used in Coq and Isabelle Exports

Only a subset of the upper level ontology are actually taken advantage of in the existing Coq and Isabelle exports. This section lists which predicates are used and wich are not.

ULO Predicates used in the Isabelle Exports [7]

`check-time` `defines` `definition` `derived` `experimental` `external-size` `function` `important` `inductive-on` `instance-of` `justifies` `name` `para` `paratype` `predicate` `primitive` `section` `sourceref` `specified-in` `specifies` `statement` `theory` `type` `unimportant` `universe` `uses`

ULO Predicates *not* used in the Isabelle Exports [7]

`action-times` `aligned-with` `alternative-for` `antonym` `automatically-proved` `axiom` `constructs` `contains` `counter-example-for` `crossrefs` `declaration` `deprecated` `docref` `example` `example-for` `file` `folder` `formalizes` `generated-by` `hypernym` `hyponym` `implementation-uses` `implementation-uses-implementation-of` `implementation-uses-interface-of` `inspired-by` `inter-statement` `interface-uses` `interface-uses-implementation-of` `interface-uses-interface-of` `internal-size` `last-checked-at` `library` `library-group` `logical` `mutual-block` `nyms` `organizational` `phrase` `physical` `proof` `proposition` `revision` `rule` `same-as` `see-also` `similar-to` `size-properties` `superseded-by` `theorem` `typedec` `uses-implementation` `uses-interface`

ULO Predicates used in the Coq Exports [8]

`axiom` `definition` `derived` `example` `file` `folder` `internal-size` `library` `library-group` `predicate` `primitive` `proposition` `revision` `statement` `theory` `type` `uses`

ULO Predicates *not* used in the Coq Exports [8]

`action-times` `aligned-with` `alternative-for` `antonym` `automatically-proved` `check-time` `constructs` `contains` `counter-example-for` `crossrefs` `declaration` `defines` `deprecated` `docref` `example-for` `experimental` `external-size` `formalizes` `function` `generated-by` `hypernym` `hyponym` `implementation-uses` `implementation-uses-implementation-of` `implementation-uses-interface-of` `important` `inductive-on` `inspired-by` `instance-of` `inter-statement` `interface-uses` `interface-uses-implementation-of` `interface-uses-interface-of` `justifies` `last-checked-at` `logical` `mutual-block` `name` `nyms` `organizational` `para` `paratype` `phrase` `physical` `proof` `rule` `same-as` `section` `see-also` `similar-to` `size-properties` `sourceref` `specified-in` `specifies` `superseded-by` `theorem` `typedec` `unimportant` `universe` `uses-implementation` `uses-interface`

References

[1] Michael Kohlhase. "Mathematical knowledge management: transcending the one-brain-barrier with theory graphs." In: *European Mathematical Society (EMS) Newsletter* 92 (2014), pp. 22–27.

[2] Katja Bercic, Michael Kohlhase, and Florian Rabe. *Towards a Heterogeneous Query Language for Mathematical Knowledge: Extended Report*. URL: `https://kwarc.info/people/mkohlhase/papers/tetrasearch.pdf` (visited on 24/06/2020).

[3] Andrea Condoluci et al. "Relational data across mathematical libraries." In: *International Conference on Intelligent Computer Mathematics*. Springer. 2019, pp. 61–76.

[4] *ulo*. MathHub. 2019. URL: `https://gl.mathhub.info/ulo/ulo` (visited on 07/07/2020).

[5] Grigoris Antoniou and Frank Van Harmelen. *A semantic web primer*. MIT press, 2004. Chap. 4, pp. 113–152.

[6] Michael Kohlhase, Florian Rabe, and Makarius Wenzel. *Making Isabelle Content Accessible in Knowledge Representation Formats*. 2020. arXiv: `2005.08884 [cs.LO]`.

[7] *Isabelle: Libraries of the Isabelle proof assistant in OMDoc/MMT representation*. MathHub. 2019. URL: `https://gl.mathhub.info/Isabelle` (visited on 10/06/2020).

[8] *XML Coq Exports*. MathHub. 2019. URL: `https://gl.mathhub.info/Coqxml` (visited on 16/06/2020).

[9] Junio C Hamano. "GIT–A stupid content tracker." In: *Proc. Ottawa Linux Sympo* 1 (2006), pp. 385–394.

[10] Mihnea Iancu et al. "System description: MathHub. info." In: *Intelligent Computer Mathematics*. Springer, 2014, pp. 431–434.

[11] *GraphDB 9.3 documentation*. Ontotext. 2020. URL: `http://graphdb.ontotext.com/documentation/free/` (visited on 16/06/2020).

[12] *RDF 1.1 XML Syntax*. W3C. 2014. URL: `https://www.w3.org/TR/rdf-syntax-grammar/` (visited on 17/08/2020).

[13] Peter Deutsch et al. *GZIP file format specification version 4.3*. Tech. rep. RFC 1952, May, 1996.

[14] L Collin and I Pavlov. *The. xz file format*. 2009. URL: `https://tukaani.org/xz/xz-file-format.txt` (visited on 17/08/2020).

[15] Tim Berners-Lee, Roy T. Fielding, and Larry M Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. Jan. 2005. DOI: `10.17487/RFC3986`. URL: `https://rfc-editor.org/rfc/rfc3986.txt` (visited on 17/08/2020).

[16] Virtuoso Open-Source Wiki. *Virtuoso Open-Source Edition*. URL: `http://vos.openlinksw.com/owiki/wiki/VOS` (visited on 27/09/2020).

[17] *Nested Repositories*. Ontotext. 2020. URL: http://graphdb.ontotext.com/documentation/standard/nested-repositories.html (visited on 23/09/2020).

[18] *What is RDF Triplestore?* Ontotext. 2020. URL: https://www.ontotext.com/knowledgehub/fundamentals/what-is-rdf-triplestore/ (visited on 17/08/2020).

[19] *Triple Store*. W3C. 2001. URL: https://www.w3.org/2001/sw/Europe/events/20031113-storage/positions/rusher.html (visited on 17/08/2020).

[20] *GraphDB Feature Comparison*. Ontotext. 2020. URL: http://graphdb.ontotext.com/documentation/free/graphdb-feature-comparison.html (visited on 17/08/2020).

[21] *SPARQL Compliance*. Ontotext. 2020. URL: http://graphdb.ontotext.com/documentation/standard/sparql-compliance.html (visited on 27/09/2020).

[22] Sean Bechhofer et al. "OWL web ontology language reference." In: *W3C recommendation* 10.02 (2004). URL: https://www.w3.org/TR/owl-ref/ (visited on 27/09/2020).

[23] *Reasoning*. Ontotext. 2020. URL: http://graphdb.ontotext.com/documentation/standard/sparql-compliance.html (visited on 27/09/2020).

[24] Eric W. Weisstein. *Transitive Closure*. URL: https://mathworld.wolfram.com/TransitiveClosure.html (visited on 27/09/2020).

[25] W3C. 2009. URL: https://www.w3.org/2009/sparql/wiki/Feature:PropertyPaths (visited on 27/09/2020).

[26] *SPARQL Query Language for RDF*. W3C. 2009. URL: https://www.w3.org/TR/rdf-sparql-query/ (visited on 10/06/2020).

[27] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Irvine, 2000.

[28] Bob DuCharme. *Learning SPARQL: querying and updating with SPARQL 1.1*. " O'Reilly Media, Inc.", 2013.

[29] *SparqlImplementations*. W3C. URL: https://www.w3.org/wiki/SparqlImplementations (visited on 06/07/2020).

[30] *package sparql*. godoc.org. 2019. URL: https://godoc.org/github.com/knakk/sparql (visited on 10/06/2020).

[31] *Eclipse rdf4j*. The Eclipse Foundation. 2020. URL: https://rdf4j.org/ (visited on 10/06/2020).

[32] *Using GraphDB with the RDF4J API*. Ontotext. 2020. URL: http://graphdb.ontotext.com/documentation/free/using-graphdb-with-the-rdf4j-api.html (visited on 10/06/2020).

[33] Peter Ansell. *ansell/rdf4j-schema-generator*. URL: https://github.com/ansell/rdf4j-schema-generator (visited on 02/07/2020).

[34] *ABOUT W3C*. W3C. URL: https://www.w3.org/Consortium/ (visited on 05/10/2020).

[35] Randall Smith. *Docker Orchestration*. Packt Publishing Ltd, 2017.

[36] Andreas Schärtl. *ULO RDF Collector*. 2020. URL: https://gitlab.cs.fau.de/kissen/ulo-storage-collect (visited on 14/09/2020).

[37] Andreas Schärtl. *Supervision Repository*. 2020. URL: https://gl.kwarc.info/supervision/schaertl_andreas/-/tree/master/experimental/compose (visited on 14/09/2020).

[38] *coq.8.9.0*. MathHub. 2019. URL: https://gl.mathhub.info/Coqxml/coq.8.9.0 (visited on 30/07/2020).

[39] *The Definitive Glossary of Higher Mathematical Jargon*. Math Vault. URL: https://mathvault.ca/math-glossary/#elementary (visited on 07/07/2020).

[40] Matthew Inglis et al. "On mathematicians' different standards when evaluating elementary proofs." In: *Topics in cognitive science* 5.2 (2013), pp. 270–282.

[41] Jeremy Avigad. "Number theory and elementary arithmetic." In: *Philosophia mathematica* 11.3 (2003), pp. 257–284.

[42] Dorian Goldfeld. "The elementary proof of the prime number theorem: An historical perspective." In: *Number Theory*. Springer, 2004, pp. 179–192.

[43] FN Maslahah, AM Abadi, et al. "Undergraduate students' difficulties in proving mathematics." In: *Journal of Physics: Conference Series*. Vol. 1320. 1. IOP Publishing. 2019, p. 012072.

[44] Muhammet Doruk and Abdullah Kaplan. "Prospective Mathematics Teachers' Difficulties in Doing Proofs and Causes of Their Struggle with Proofs." In: *Online Submission* 10.2 (2015), pp. 315–328.

[45] Matti Järvisalo et al. "Relating proof complexity measures and practical hardness of SAT." In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2012, pp. 316–331. URL: https://www.cs.helsinki.fi/u/mjarvisa/papers/jarvisalo-matsliah-nordstrom-zivny.cp12.pdf (visited on 30/07/2020).

[46] Junkuo Cao, Weihua Wang, and Yuanzhong Shu. "Comparison of pagination algorithms based-on large data sets." In: *International Symposium on Information and Automation*. Springer. 2010, pp. 384–389.

[47] Tom Wiesing, Michael Kohlhase, and Florian Rabe. "Virtual theories–a uniform interface to mathematical knowledge bases." In: *International Conference on Mathematical Aspects of Computer and Information Sciences*. Springer. 2017, pp. 243–257. URL: https://kwarc.info/people/frabe/Research/WKR_virtual_17.pdf (visited on 09/07/2020).

[48] Paul Nijjar. "An attempt to automate np-hardness reductions via SO∃ logic." MA thesis. University of Waterloo, 2004.

[49] John Kunze and Thomas Baker. *The Dublin core metadata element set*. Tech. rep. RFC 5013, August, 2007.

[50] *DCMI Metadata expressed in RDF Schema Language*. Dublin Core Metadata Initiative. URL: `https://www.dublincore.org/schemas/rdfs/` (visited on 30/06/2020).

[51] *Rosetta Code*. URL: `https://www.rosettacode.org/wiki/Rosetta_Code` (visited on 30/09/2020).

[52] Iztok Fister Jr, Iztok Fister, and Matjaž Perc. "Toward the discovery of citation cartels in citation networks." In: *Frontiers in Physics* 4 (2016), p. 49.

[53] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, And Tools: Second Edition*. 2007.